Trimester 1 2020

Code-onavirus

D.O.T (Disease Outbreak Tracker)
System Testing Report

Written by: Kenvin Yu, Victor Liu, Eric Tan,
Ezra Eyaru and Evan Lee

Testing Environment

In order to ensure the system would meet the functionality and reliability required, the testing environment involved both local tests as well as tests utilising the server and API endpoint hosted by the server.

Local tests primarily focused on the correctness and logic of code modules that made up things such as the scraper, parsing scraped data in the database and correctness of API calls. Through testing these files locally, unrelated variables are easily removed resulting in being able to more easily pinpoint exactly where errors are in code ensuring that the code is functionally correct.

Testing in the online environment was mostly to ensure that the endpoint queries were all functioning once all the other code in the back end of the system was verified to be correct. This also includes factors irrelevant to the functionality of the code such as success and error code return values from the API endpoint. Again, this eliminated potential points of error resulting in easier troubleshooting. Overall, the aim of the testing environment setup was to find bugs quickly, ensuring a system that would be reliable through consistently returning correct information to the user.

Tools Used

We used Python's Pytest module as our choice of testing tool. The Flask web server we are using also offers easy integration with the Pytest module by providing a testing client as an environment to facilitate the black-box testing for our RESTful API web service. The built-in unit test module for Python is also too verbose in agile development, whereas Pytest has the advantage of having simpler test creation and offers more flexibility for running our tests. To assist in our testing, we used JSON files to represent sample inputs and expected outputs for our tests, which were simple to process using Python's JSON library.

Limitations

There were several limitations in our testing process. As WHO has more than 2800 articles dating back to 1996, it would require many man-hours to test our program with every single one as each article would need to be understood by a person, who then has to generate the expected output. As such, we have selected different types of articles over all years for our sample inputs to ensure that we accommodate for stylistic or formatting changes that occur over time. For white-box testing of individual modules and functions, there are more than 80 test cases or articles that were manually examined. This translates to less than 2% coverage of all articles. Another limitation was the black-box testing of our RESTful API web service as the results of our API calls are dependent on the contents of our database, which is populated by scraping and parsing. When the underlying code for scraping or parsing articles was changed, it was necessary to repopulate or update the database, which consumed a considerable amount of time due to the large quantity of articles and complex parsing required. Also, due to the agile nature of our development, there were several utility or helper functions that were absent from being tested due to uncertainty of them being used or being impractical to test standalone. Examples of such utility functions include functions that solely interface with the database such as ones that populate the database with scraped article data.

Testing Methods

Black Box Testing

The black-box testing of our RESTful API was simplified by Flask's integration with Pytest, allowing us to set up a test client that works with Pytest's configuration fixture for all API-based tests. In the process of black-box testing, we first prioritised testing invalid API calls to confirm that our API correctly performs input validation and generates relevant error codes for the end-user. These calls include incorrect URLs for resources, incorrect methods and invalid request payload bodies, where the last can occur for a variety of reasons such as formatting and date ordering. Within these tests, the response was inspected for the expected HTTP status codes as well as appropriate error messages detailing the issues with the API call. After ensuring valid input, we tested the API's ability to filter queries made to valid endpoints by date range, key terms, locations and combinations of such. For these test cases, the query parameters were carefully chosen to include only specific articles such that we could determine that the filters were functional. Of course, the JSON output from these test cases were compared with JSON result files to ensure that they contained the expected article details for each query. For black-box testing, our API web server satisfied 100% of all test cases.

Testing Methods

White Box Testing

For white-box testing, we focused on testing functions that were used for parsing and generating some form of report data from gathered articles. Examples of such functions include text processing and parsing to determine event dates, key terms, locations, diseases and syndromes within an article. As aforementioned in our previous discussion of testing limitations, we have selected articles across all available years as test cases to ensure our parsing functions are flexible enough to cover a wide range of language and formatting styles. Similarly, for testing disease parsing we have selected a wide range of articles encompassing every disease present in the disease_list JSON file, ensuring all diseases are properly recognised and parsed from text. We supplied the input test data for each case using JSON files provided by our web scraper. For each test, the accompanying article is manually inspected by a human to derive the expected output for the test. The expected output for each case can be some form of JSON object, or it could simply be a string to match, entirely dependent on the output of the code run.

Improvements From Testing

We employed a cycle of making tests and implementation to ensure that we iteratively improved our processing to accommodate for the widest range of test cases. An example of such is the date parsing module which passes a total of 48 out of 49 test cases. While not perfect, we can be confident that date parsing will produce a relevant result for about 98% of all articles since our test cases were sourced over all available years. Also with regard to the input payload validation for API calls, when new invalid tests were created that failed to produce the expected output, the underlying implementation of the parsing and verification logic was promptly rectified.